



**YosysHQ**

# New Open Source Design Verification Tools from YosysHQ

N. Engelhardt

# Contents

- Yosys and Yosys-based Tools
  - Yosys
  - SBY formal property checker
  - MCY mutation coverage
- New Developments
  - EQY equivalence checker
    - Example: Equivalence Checking an
  - SCY cover sequence generator
- Get the Tools
- Questions

# Yosys and Yosys-based Tools

# Yosys – a swiss army knife for netlists

- Open Source project started in 2012 by Claire Wolf
  - Originally a synthesis tool for an academic CGRA
  - Grew in capabilities and language support
- Now a tool that can be applied in many different contexts, anytime you need to transform netlists
  - Used as "glue" between many third-party tools
  - And for architecture exploration
  - But also a fully-fledged synthesis tool, used e.g. in the OpenLANE ASIC flow and by some FPGA vendors
    - Much of the heavy lifting is done by abc - stay for the next talk to learn details!

# Yosys – input and output formats

- Input formats:

- Verilog
- JSON
- Aiger
- Blif
- Liberty
- VHDL (GHDL plugin)

- Commercial Edition adds:

- SystemVerilog
- SystemVerilog Assertions
- VHDL

- Output formats

- Verilog
- JSON
- Blif
- EDIF
- FIRRTL
- Aiger
- SMT2
- BTOR2
- C++ (simulation)
- Truth table

Full List of Commands:

[https://yosyshq.readthedocs.io/projects/yosys/en/latest/cmd\\_ref.html](https://yosyshq.readthedocs.io/projects/yosys/en/latest/cmd_ref.html)

# Yosys – Transformations

- General structure of Yosys-based flows
  - Run commands to read and elaborate the design
  - Run coarse-grain optimization commands
  - (Optional: Map to a fine-grain representation and run fine-grain optimizations)
  - Run back-end command to write design to output file
- Creating custom functionality using existing passes
  - Yosys has a rich set of commands to
    - Elaborate, simplify, infer, synthesize, technology map, simulate, ...
    - See [https://yosyshq.readthedocs.io/projects/yosys/en/latest/cmd\\_ref.html](https://yosyshq.readthedocs.io/projects/yosys/en/latest/cmd_ref.html)
  - One can get very far with creative selections of design elements and combinations of passes
- Creating custom functionality using custom passes
  - Techmap rules (module substitution - verilog file with special names)
  - Plugins (C++) can add custom passes with the same API used by internal passes
  - Pattern Matcher Generator - find subgraphs and modify/replace them

# SBY – formal property checking with Yosys

- Frontend for formal flows
  - Allows easy use of SystemVerilog `assume()`, `assert()`, `cover()` statements
    - Complex SVA properties/sequences are supported with the commercial version
  - SBY has modes for bounded and unbounded proofs
    - Support for different unbounded proof methods (k-induction, pdr/ic3)
- Automates the steps for running formal proofs with Yosys
  - Yosys translation of design to formal problem formats (SMT2, BTOR2, Aiger...)
  - Running solvers to find a set of signal values responding to the problem (or not)
    - Allows using many solvers being developed by researchers
  - Using Yosys to translate the set of variable assignments back into a VCD trace
- Myriad of different input/output formats “under the hood”
  - SBY provides a uniform interface for a wide range of solvers, hiding those differences.
- Example projects:
  - riscv-formal: formally verify ISA compliance (rv32imc/rv64imc) <https://github.com/YosysHQ/riscv-formal/>
  - AXI4 formal verification IP (requires SVA support) <https://github.com/YosysHQ-GmbH/SVA-AXI4-FVIP>

# SBY – formal property checking with Yosys

```
module demo (  
    input clk,  
    output reg [5:0] counter  
);  
    initial counter = 0;  
  
    always @(posedge clk) begin  
        if (counter == 15)  
            counter <= 0;  
        else  
            counter <= counter + 1'b1;  
        end  
  
    `ifdef FORMAL  
        always @(posedge clk) begin  
            assert (counter < 32);  
        end  
    `endif  
endmodule
```

```
[options]  
mode bmc  
depth 3  
  
[engines]  
smtbmc  
  
[script]  
read -formal demo.sv  
prep -top demo  
  
[files]  
demo.sv
```

```
> sby -f demo.sby  
SBY 18:06:15 [demo] Removing directory '/Users/nak/Source/sby/docs/examples/quic  
kstart/demo'.  
SBY 18:06:15 [demo] Copy '/Users/nak/Source/sby/docs/examples/quickstart/demo.sv  
' to '/Users/nak/Source/sby/docs/examples/quickstart/demo/src/demo.sv'.  
SBY 18:06:15 [demo] engine_0: smtbmc  
SBY 18:06:15 [demo] base: starting process "cd demo/src; yosys -ql ../model/desi  
gn.log ../model/design.yes"  
SBY 18:06:15 [demo] base: finished (returncode=0)  
SBY 18:06:15 [demo] prep: starting process "cd demo/model; yosys -ql design_prep  
.log design_prep.yes"  
SBY 18:06:15 [demo] prep: finished (returncode=0)  
SBY 18:06:15 [demo] smt2: starting process "cd demo/model; yosys -ql design_smt2  
.log design_smt2.yes"  
SBY 18:06:15 [demo] smt2: finished (returncode=0)  
SBY 18:06:15 [demo] engine_0: starting process "cd demo; yosys-smtbmc --presat -  
-unroll --noprogess -t 3 --append 0 --dump-vcd engine_0/trace.vcd --dump-yw en  
gine_0/trace.yw --dump-vlogtb engine_0/trace_tb.v --dump-smtc engine_0/trace.smt  
c model/design_smt2.smt2"  
SBY 18:06:15 [demo] engine_0: ## 0:00:00 Solver: yices  
SBY 18:06:15 [demo] engine_0: ## 0:00:00 Checking assumptions in step 0..  
SBY 18:06:15 [demo] engine_0: ## 0:00:00 Checking assertions in step 0..  
SBY 18:06:15 [demo] engine_0: ## 0:00:00 Checking assumptions in step 1..  
SBY 18:06:15 [demo] engine_0: ## 0:00:00 Checking assertions in step 1..  
SBY 18:06:15 [demo] engine_0: ## 0:00:00 Checking assumptions in step 2..  
SBY 18:06:15 [demo] engine_0: ## 0:00:00 Checking assertions in step 2..  
SBY 18:06:15 [demo] engine_0: ## 0:00:00 Status: passed  
SBY 18:06:15 [demo] engine_0: finished (returncode=0)  
SBY 18:06:15 [demo] engine_0: Status returned by engine: pass  
SBY 18:06:15 [demo] summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:00 (0)  
SBY 18:06:15 [demo] summary: Elapsed process time [H:MM:SS (secs)]: 0:00:00 (0)  
SBY 18:06:15 [demo] summary: engine_0 (smtbmc) returned pass  
SBY 18:06:15 [demo] summary: engine_0 did not produce any traces  
SBY 18:06:15 [demo] DONE (PASS, rc=0)
```



# MCY – Mutation Coverage with Yosys

- Mutation coverage is a coverage metric for testbenches
  - Solves the issue of false negatives that is inherent to execution/branch coverage
  - Introduce modifications to the DUT and see if each modification causes the tests to fail
  - Yosys modifies the netlist and outputs a modified module to instantiate in the testbench
  - Works with any self-checking test environment that accepts a synthesized DUT
- Main Problem with Mutation coverage: False Positives
  - Some mutations don't violate the design spec, so it's fine for the test bench not to fail for them
- MCY Solution: Filter False Positives with formal equivalence checks
  - Create a miter circuit with mutated and non-mutated design, to let the formal method investigate the functional change introduced by such a mutation
  - Optional: Write properties taking into account when differences are relevant
    - e.g., only compare data if data\_valid is high
    - Much easier than writing formal properties about the expected value of data
  - Mutations that create no relevant functional change are discarded automatically
  - Can run the formal checks in SBY, or interface with formal tools from other vendors

# MCY – Mutation Coverage with Yosys

```
[options]
size 1000
tags COVERED UNCOVERED NOCHANGE EQGAP FMONLY

[script]
read -sv bitcnt.v
prep -top bitcnt

[files]
bitcnt.v

[logic]
use_formal = False

tb_okay = (result("test_sim") == "PASS")
eq_okay = (result("test_eq") == "PASS")

if tb_okay and use_formal:
    tb_okay = (result("test_fm") == "PASS")
    if not tb_okay:
        tag("FMONLY")

if tb_okay and not eq_okay:
    tag("UNCOVERED")
elif not tb_okay and not eq_okay:
    tag("COVERED")
elif tb_okay and eq_okay:
    tag("NOCHANGE")
elif not tb_okay and eq_okay:
    tag("EQGAP")
else:
    assert 0

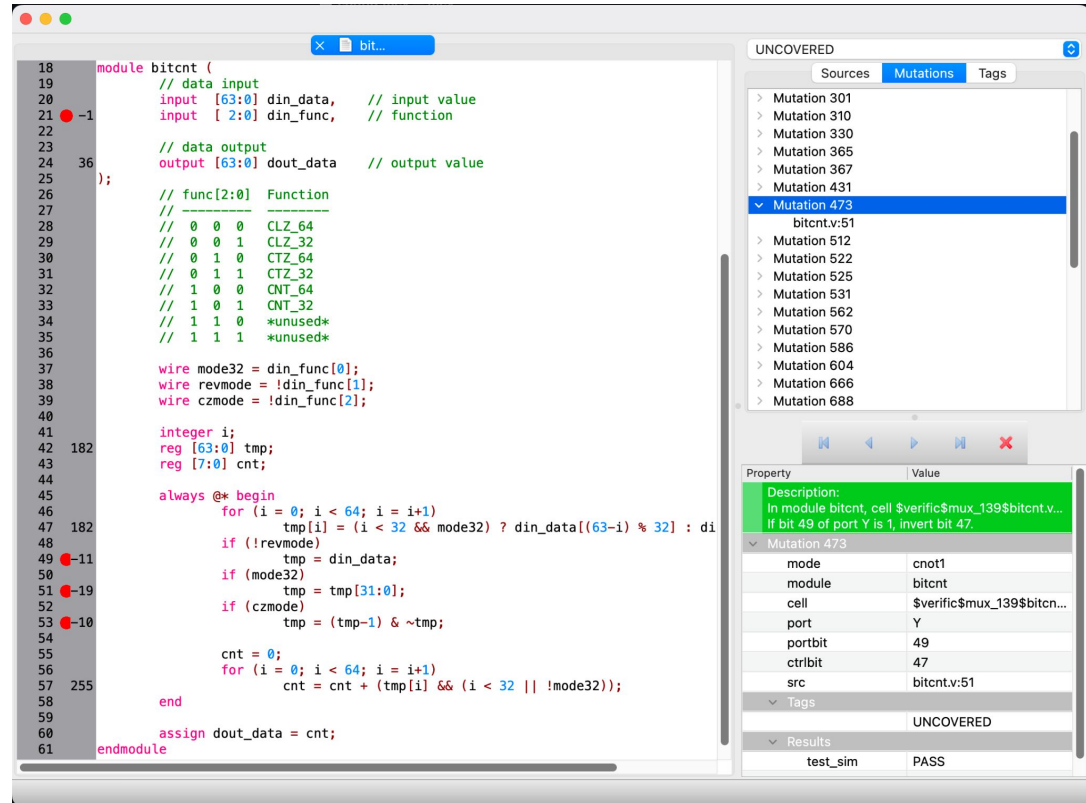
[report]
if tags("EQGAP"):
    print("Found %d mutations exposing a formal equivalence gap!" % tags("EQGAP"))
if tags("COVERED")+tags("UNCOVERED"):
    print("Coverage: %.2f%%" % (100.0*tags("COVERED")/(tags("COVERED")+tags("UNCOVERED"))))

[test test_sim]
expect PASS FAIL
run bash $PRJDIR/test_sim.sh

[test test_eq]
expect PASS FAIL
run bash $PRJDIR/test_eq.sh

[test test_fm]
expect PASS FAIL
run bash $PRJDIR/test_fm.sh

> mcy status
Database contains 2000 cached results.
Database contains 938 cached "FAIL" results for "test_eq".
Database contains 62 cached "PASS" results for "test_eq".
Database contains 898 cached "FAIL" results for "test_sim".
Database contains 102 cached "PASS" results for "test_sim".
Tagged 898 mutations as "COVERED".
Tagged 62 mutations as "NOCHANGE".
Tagged 40 mutations as "UNCOVERED".
Coverage: 95.74%
```



The screenshot shows the Yosys GUI with a Verilog module named 'bitcnt' open in the editor. The module has three inputs: 'din\_data' (63:0), 'din\_func' (2:0), and 'dout\_data' (63:0). It has two outputs: 'dout\_data' (63:0) and 'cnt' (7:0). The module implements a counter that increments based on the 'din\_func' input. The 'cnt' output is assigned to 'dout\_data'.

On the right side of the GUI, there is a panel titled 'UNCOVERED' showing a list of mutations. The 'Mutation 473' is selected, and its details are shown in the 'Property' table below.

Property	Value
<b>Description:</b> In module bitcnt, cell \$verific\$mux_139\$bitcnt.v... If bit 49 of port Y is 1, invert bit 47.	
Mutation 473	
mode	cnot1
module	bitcnt
cell	\$verific\$mux_139\$bitcnt...
port	Y
portbit	49
ctrlbit	47
src	bitcnt.v:51
Tags	UNCOVERED
Results	
test_sim	PASS

# New Developments

# EQY – Equivalence Checking with Yosys

- Our brand new equivalence checking tool
- Identifies matching points in two designs
  - Then partitions the design into smaller pieces that can be checked independently
  - Scales much better than using SBY on a miter circuit
  - Much easier to identify parts of design that cause scaling issues
- Application domains
  - Ensure post-synthesis netlist is the same as input design
  - Check that a non-functional change does not change the behavior of the design
- Example/tutorial projects included with the tool
  - Verification of a design change in ALU/shifter architecture in NERV RISC-V Processor
  - Verification of Xilinx Vivado synthesis output for PicoRV32 processor design

# Equivalence Checking an OpenLANE/SKY130 Netlist

- Starting from a small example design (`spm`), walking through the process for using EQY to compare the netlist produced by OpenLANE to the original RTL

## Files:

- `spm.v` : the original source code
- `spm.nl.v` : the netlist, found in `results/final/verilog`
- `primitives.v` and `sky130_fd_sc_hd.v` : the cell library simulation models from the SkyWater PDK.

# Adapting the simulation models

- The simulation library provided with the SkyWater PDK can't be directly read in with yosys (e.g., some ``ifdef` branches have unsupported syntax).
- My colleague Jannis wrote a small script that makes a few modifications:
  - Resolves ``ifdefs` (with an implicit ``FUNCTIONAL` define) and removes ``UNIT_DELAY`. (This isn't a full Verilog preprocessor, but it's enough to handle the PDK Verilog files.)
  - Adds `(* noblackbox *)` attributes to all modules, as the PDK contains some modules without logic.
  - Replaces `pullup` and `pulldown` primitives which Yosys doesn't support with `mod_pullup` and `mod_pulldown` instances.
  - Replaces `pwrgood` with primitives that assume the power is always good.
  - Automatically replaces combinational UDPs with a `casez`-based module implementation.
  - Replaces the few remaining stateful UDPs with manually written synthesizable modules.

## Setting up the EQY Project (`spm.eqy`)

- `[gold]`: yosys commands that run only on original RTL
- `[gate]`: yosys commands that run only on netlist
- `[script]`: yosys commands that run on both
- `[strategy ...]`: proof methods to try on partitions of the design
  - 'sat' is the yosys-internal sat command
  - 'sby' creates an SBY project
  - tried in order, the second strategy is used if the first is inconclusive
  - we are planning to add more strategies

```
[gold]
read_verilog -formal spm.v
```

```
[gate]
exec -- python3 formal_pdk_proc.py
primitives.v sky130_fd_sc_hd.v -o
spm/formal_pdk.v
read -sv spm/formal_pdk.v spm.nl.v
```

```
[script]
hierarchy -check -top spm
prep
async2sync
```

```
[strategy sat]
use sat
depth 2
```

```
[strategy sby]
use sby
depth 2
engine smtbmc bitwuzla
```

## Running EQY

- `eqy -f spm.eqy`
- runs the specified scripts to read in the gold and gate designs
- partially flattens on either side until the hierarchy of both matches
- matches wires on both sides to find equivalent points (by name)
- fragments the design into small pieces along equivalent points
- groups fragments together to form partitions
- proves each partition equivalent by trying several strategies
  - this is sequential equivalence, including state, e.g. using k-induction
- concludes about the design overall.



# Additional challenges in more complex designs

- Guiding the tool in finding equivalent points
  - If a different synthesis tool is used, the net bearing the same name in the gate may not have the same semantics
    - [match ...] section
  - FSM recoding
    - [recode ...] section
- Guiding the tool in fragmenting and partitioning the design
  - Sometimes fragments and partitions get too small
  - If signals are related, certain portions of the design are only equivalent if this relationship is known (e.g.  $a \text{ xor } b$  is equivalent to 1 if you know that  $a = \sim b$ )
    - [collect ...] and [partition ...] sections

# Running EQY

```
eqy -f spm.eqy
EQY 11:44:43 [spm] read_gold: starting process "yosys -ql spm/gold.log spm/gold.y"
EQY 11:44:43 [spm] read_gold: finished (returncode=0)
EQY 11:44:43 [spm] read_gate: starting process "yosys -ql spm/gate.log spm/gate.y"
EQY 11:44:43 [spm] read_gate: finished (returncode=0)
EQY 11:44:43 [spm] combine: starting process "yosys -ql spm/combine.log spm/combine.y"
EQY 11:44:43 [spm] combine: finished (returncode=0)
EQY 11:44:44 [spm] partition: starting process "cd spm; yosys -ql partition.log partition.y"
EQY 11:44:44 [spm] partition: finished (returncode=0)
EQY 11:44:44 [spm] run: starting process "make -C spm -f strategies.mk"
EQY 11:44:44 [spm] run: make[1]: Entering directory '/home/nak/Work/eqy/examples/spm/spm'
EQY 11:44:44 [spm] run: Running strategy 'sat' on 'spm.csa0.hsum2'..
EQY 11:44:45 [spm] run: Proved equivalence of partition 'spm.csa0.hsum2' using strategy 'sat'
EQY 11:44:45 [spm] run: Running strategy 'sat' on 'spm.csa0.sc'..

EQY 11:44:49 [spm] Successfully proved equivalence of partition spm.genblk1.11.csa.y
EQY 11:44:49 [spm] Successfully proved equivalence of partition spm.genblk1.11.csa.sc
EQY 11:44:49 [spm] Successfully proved equivalence of partition spm.genblk1.11.csa.hsum2
EQY 11:44:49 [spm] Successfully proved equivalence of partition spm.genblk1.10.csa.y
EQY 11:44:49 [spm] Successfully proved equivalence of partition spm.genblk1.10.csa.sum
EQY 11:44:49 [spm] Successfully proved equivalence of partition spm.genblk1.10.csa.sc
EQY 11:44:49 [spm] Successfully proved equivalence of partition spm.genblk1.10.csa.hsum2
EQY 11:44:49 [spm] Successfully proved equivalence of partition spm.csa0.y
EQY 11:44:49 [spm] Successfully proved equivalence of partition spm.csa0.sc
EQY 11:44:49 [spm] Successfully proved equivalence of partition spm.csa0.hsum2
EQY 11:44:49 [spm] Successfully proved designs equivalent
EQY 11:44:49 [spm] summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:05 (5)
EQY 11:44:49 [spm] summary: Elapsed process time [H:MM:SS (secs)]: 0:00:05 (5)
EQY 11:44:49 [spm] DONE (PASS, rc=0)
```

# SCY – Sequence of Covers with Yosys

- Sneak peek at our next development:

A formal methodology and tool for generating long cover traces for large designs, based on “checkpoint” cover properties, that the tool eagerly solves one-by-one, using the final state of one property as the initial state of the next.

- Example Applications:

- Creating formal cover traces for complex bus interactions on large SoC designs.
- Using formal tools to create assembler programs to put a processor in difficult to reach states of its state-space

# Data-Flow Properties

- With SCY we will also introduce a methodology for formal data-flow properties, for example:
  - Cover a trace that shows `top.Bus.ComponentA.DOUT_VALID` and `top.Bus.ComponentA.DOUT_READY` active in cycle  $t_1$ ,
  - and `top.Bus.ComponentB.DIN_VALID` and `top.Bus.ComponentB.DIN_READY` active in cycle  $t_2$ ,
  - and `top.Bus.ComponentB.DIN_DATA` at  $t_2$  is a function of `top.Bus.ComponentA.DOUT_DATA` at  $t_1$ .
- Where “is a function of” means we can show data-flow (of a configurable kind) from the input to the output.
- This functionality is especially useful for the kind of properties we are building SCY for, but will be made available in all our formal flows.

# Get the Tools

## Try it out!

- Download nightly builds of the OSS CAD Suite
  - <https://github.com/YosysHQ/oss-cad-suite-build/releases/latest>
  - Includes Yosys, SBY, MCY, all dependencies, supported solvers, GHDL plugin (linux only)
  - Also nextpnr, Amaranth, cocotb, ...
- Documentation: <https://yosyshq.readthedocs.io/en/latest/>
- Ask me for an evaluation license to the commercial Tabby CAD Suite
  - Email [contact@yosyshq.com](mailto:contact@yosyshq.com) or tick a box on <https://www.yosyshq.com/contact>

# Thank You

- to our excellent dev team
- in particular to the colleagues who helped with this presentation:
  - Claire Wolf
  - Jannis Harder
  - Matt Venn
- to contributors on github
- to you for listening!

# Q&A